Where Tests Fall Short: Empirically Analyzing Oracle Gaps in Covered Code

Megan Maton
University of Sheffield

Gregory M. Kapfhammer
Allegheny College

Phil McMinn
University of Sheffield

Abstract—*Background:* **Developers often rely on statement coverage to assess test suite quality. However, statement coverage alone may only lead to 10% fault detection, necessitating more rigorous approaches. While mutation testing is effective, its execution and human analysis costs remain high. Identifying covered statements that are not checked by oracles (e.g., assertions) offers a cost-effective alternative; however, the lack of empirical evidence for selecting the appropriate Oracle Gap Calculation Approach (OGCA) prevents developers from making informed choices.** *Aims:* **This knowledge-seeking study compares oracle gap characteristics determined by different OGCAs to assist developers in choosing the most valuable approach for their use cases.** *Method:* **Using mixed-method empirical analysis, we conduct an in-depth evaluation of the oracle gaps produced using three OGCAs: Checked Coverage using a Dynamic Slicer (CC_{DS}), Checked Coverage using an Observational Slicer (CC_{OS}), and Pseudo-Tested Statement Identification (PTSI). Across 30 Java classes from six open-source projects, we report on a quantitative evaluation of gap prominence, distribution, fault detection correlation and execution times, as well as results from a qualitative manual inspection of the statement types found in the oracle gaps.** *Results:* **The qualitative analysis showed data-loading statements, iteration statements and output updates to be most prominent in the oracle gaps. PTSI identified the oracle gaps with the lowest median mutation score (0.32), highlighting areas requiring more fault detection improvement compared to CC_{DS} (0.76) and CC_{OS} (0.50). PTSI also had the shortest median execution time (19.9 seconds), far quicker than both CC_{DS} (273.2 seconds) and CC_{OS} (5957.1 seconds).** *Conclusions:* **PTSI quickly reveals the priority testing areas for improved fault detection, making it an effective OGCA for developers to identify where tests fall short.**

I. Introduction

Due to insufficient time allocation and limited recognition of testing effort [1], developers understandably struggle with motivation to test their code. Meeting metric targets, such as code coverage, gives developers a clear goal. But with limited time for testing, focusing on maximizing only the scores of these metrics may result in other test weaknesses emerging.

Code coverage (i.e., statement or branch coverage) is the most common metric for assessing test quality, with its advantages demonstrated in studies from Google and IBM [2], [3]. Yet despite widespread use, code coverage has many well-known limitations. For instance, writing a test suite to achieve statement coverage can result in only 10% fault detection [4].

The current state-of-the-art technique for assessing fault detection is mutation testing [5], [6], [7]. Mutation testing places synthetic faults (i.e., mutants) into program code to identify whether the test suite detects them, thus helping developers establish missing test oracles. Despite being an effective simulation of real fault detection, mutation testing, in its standard interpretation, remains impractical due to the considerable computational expense and the manual effort required to identify equivalent mutants [8], [9], [10], [11].

Recent evidence has found that identifying *oracle gaps*, that is, code that is executed by tests (i.e., covered) but does not cause a test case to pass or fail, may be a cost-effective way to find where tests fall short. Intuitively, code in these gaps is not well tested by the test suite [12], [13]. Existing oracle gap calculation approaches (OGCAs), such as host checked coverage, extreme mutation testing (XMT) and identifying pseudo-tested statements, highlight covered code that is unchecked by an oracle (e.g., a test assertion) to provide developers with clear testing direction without the expense of equivalent mutant analysis and execution times [12], [13], [14].

Whilst each OGCA has been given varying attention, there exists no empirical evidence to guide developers in choosing the appropriate one [12], [13], [14], [15], [16], [17]. As such, developers remain uninformed regarding suitable oracle gap sizes; their calculation costs; the statement types within the gaps, and the mutation scores of the statements in the oracle gaps. For developers, answering these questions is critical. An empty gap offers no testing direction, while an overly large oracle gap is not cost-effective for developers to address. Furthermore, if an OGCA places irrelevant statements in the oracle gap, then the developer may waste time strengthening tests for unimportant code. Conversely, low mutation scores across oracle gap statements highlight a test suite with low fault detection, necessitating targeted test improvements.

This paper performs a knowledge-seeking study to analyze and compare the statements within the oracle gaps identified by three OGCAs: checked coverage using a dynamic slicer (CC_{DS}); checked coverage using an observational slicer (CC_{OS}) and pseudo-tested statement identification (PTSI) for 30 Java classes across six open-source projects, including libraries and SDKs, ranging in program size and test coverage. Supporting this study, we present a generalized oracle gap definition enabling comparison between them. This paper's empirical results will help developers pick the right OGCA to make the most effective use of their limited testing time.

We quantitatively evaluate the different oracle gaps for their prominence, statement distribution, correlation with fault detection, and execution times. A manual inspection with negotiated agreement revealed data loading statements, iteration

979-8-3315-9147-2/25/$31.00 ©2025 IEEE

Motivating Example

□ Inconsequent to assertion pass/fail ▨ Contributes to assertion pass/fail

D O R — *Oracle-based adequacy criteria*

```
 1 □ ▨ □ public String createXmlTag(String name,
 2 □ ▨ □                             String content) {
 3 ▨ □ □   if (name == null || name.trim().isEmpty()) {
 4 □ □ □       System.err.println("/* ... */");
 5 □ □ □       return "";
 6 □ □ □   }
 7 ▨ ▨ □   String s = "<" + name + ">";
 8 ▨ □ □   s += content;
 9 ▨ ▨ ▨   s += "</" + name + ">";
10 ▨ ▨ ▨   return s;
11 □ ▨ □ }
12
13       @Test
14       public void test_checkEndTag() {
15           String tagName = "message";
16           String tagContent = "Hello, world!";
17           String result = createXmlTag(tagName, tagContent);
18           assertTrue(result.contains("</message>"));
19       }
```

Listing 1. Motivating example using prepared Java method `createXmlTag` and its accompanying JUnit test. Column headers: **D**: Dynamic Slice; **O**: Observational Slice; **R**: Required (not pseudo-tested). The highlighted text is "covered" under statement coverage. Statements that are covered but do not contribute to an assertion passing or failing are in the oracle gap.

statements, and output updates to be the most prominent statement types and patterns. Generally, the differences between the statements in the oracle gaps indicate that the OGCAs cannot be used interchangeably. PTSI highlighted code locations with the lowest median mutation scores of 0.32, compared to CC_{DS} (0.76) and CC_{OS} (0.50), therefore revealing priority testing areas for improving fault detection. PTSI also had the shortest median execution time of 19.9 seconds, compared to 273.2 seconds (CC_{DS}) and 5957.1 seconds (CC_{OS}), suggesting it could be a good initial test suite analysis approach to help developers to rapidly identify weak testing areas.

The contributions of this paper are, therefore, as follows:

1) A generalized definition of the oracle gap.
2) A quantitative evaluation of the oracle gap prominence, distribution, fault detection rate, and execution times using checked coverage with dynamic and observational slicing, and pseudo-tested statement identification.
3) A qualitative manual inspection of code patterns identified as ineffectual by each technique.

A complete replication package is available at [18], containing all tools, execution data, manual inspection decisions and project links to enable replication and extension of this work.

II. Oracle Gaps

The motivating example in Listing 1 illustrates how using different techniques to evaluate an oracle, in this case, the assertion on Line 18, can yield conflicting oracle gaps. The example test suite contains a single test case to check the Java method `createXmlTag`, where the highlighted gray lines represent the statements that the test suite covers. The columns of boxes represent the statements on a given line that cause the assertion on Line 18 to pass or fail, as decided by the three techniques. For example, column **D** represents a dynamic slice from the assertion, for which the slice contains the statements on Lines 3, 7, 8, 9 and 10 (marked by ▨). Conversely, column **O**, which shows an observational slice, contains Lines 1, 2, 7, 9, 10 and 11. Interestingly, the statements on Lines 3 and 8 are not on the observational slice (marked by □) despite being covered (i.e., highlighted gray). As such, these statements are in the oracle gap between statement coverage and the observational slice. To remove the statements from the gap, a developer should write further tests evaluating the tag content and invalid inputs. The dynamic slice, however, contains all of the covered statements and, therefore, has an empty oracle gap, implying no further tests are needed. Column **R** shows required statements, which are covered statements that, when individually deleted or set to a different value, will cause a test to fail (marked by ▨). In this example, required statements create the largest oracle gap (i.e, statements on lines 3, 7 and 8). The discrepancy between the oracle gaps leaves developers in the dark, unsure which code is tested and which oracle gaps are actually useful to improve their testing.

A. Oracle Gaps

Test oracles (i.e., the JUnit assertion on Line 18 in Listing 1) determine whether a test case passes or fails and are vital in assessing expected program behavior [19], [20]. Since the number of assertions demonstrates a strong positive correlation with test suite effectiveness (i.e., fault finding), identifying missing oracles is core to test suite improvement [21].

Recent studies have taken a promising approach to identifying missing assertions for covered code. Each of these Oracle Gap Calculation Approaches (OGCAs) identifies covered code that does not fulfill a secondary oracle assessment criterion as shown in Listing 1. Hossain et al. defined the coverage gap as elements executed by tests but unobserved by a test oracle [12]. In extreme mutation testing (XMT), pseudo-tested methods are executed by the test suite, yet the method body is removable without causing test failures [14], [15]. Maton et al. extended this to the statement level, finding pseudo-tested statements in non-pseudo-tested methods [13]. For this study, we generalize this as an oracle gap as defined below.

Definition II.1 (Generalized Oracle Gap)**.** All covered statements that do not directly cause a test oracle to pass or fail.

This definition divides covered code into two categories:

Definition II.2 (Effectual Statement)**.** A covered statement that directly causes a test oracle to pass or fail.

Definition II.3 (Ineffectual Statement)**.** A covered statement that does **not** directly cause a test oracle to pass or fail.

This section explains how checked coverage using dynamic slicing (CC_{DS}), checked coverage using observational slicing (CC_{OS}), and pseudo-tested statement identification (PTSI) fit under these classifications. We refer to their respective oracle gaps using $gap(\text{CC}_{DS})$, $gap(\text{CC}_{OS})$ and $gap(\text{PTSI})$.

B. Checked Coverage using Dynamic Slicing (CC_{DS})

Checked coverage (CC) uses a dynamic slice (DS) from an assertion to identify the statements that cause an oracle to pass or fail [22], [23]. A dynamic slice comprises the transitive

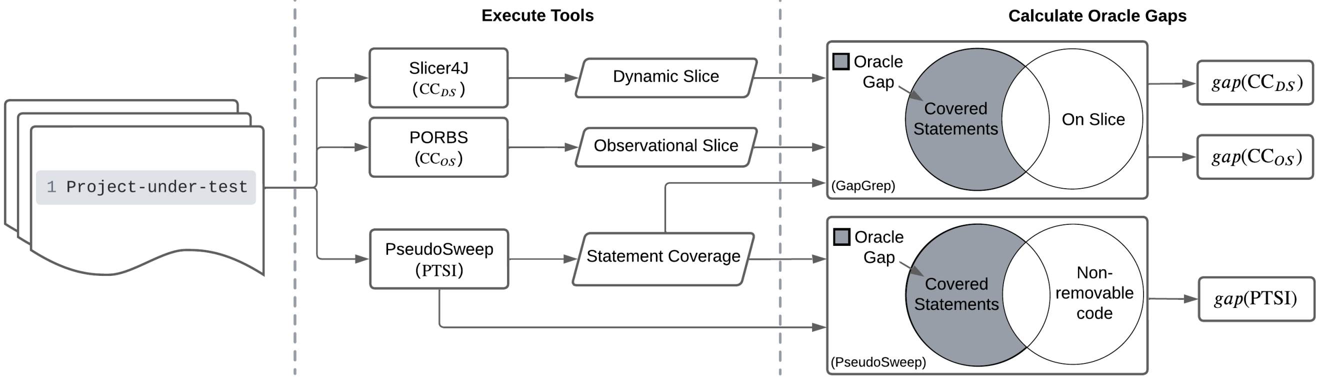

Fig. 1. Identifying Oracle Gaps with GapGrep: Oracle gaps are calculated by combining the outputs of Slicer4J (CC_{DS}), and PORBs (CC_{OS}) and statement coverage. PseudoSweep (PTSI) calculates its oracle gap (i.e., the pseudo-tested statements) internally; however, we visualize it separately for clarity.

dynamic data and control dependencies contributing to the slicing criterion (e.g., a variable value for a given input) [24].

Definition II.4 (Checked Coverage using Dynamic Slicing)**.** Checked Coverage using Dynamic Slicing requires each sliceable statement to be on at least one dynamic backward slice from an explicit test suite check.

The OGCA using Checked Coverage using Dynamic Slicing is labeled CC_{DS}. The $gap(CC_{DS})$ is the set of covered statements that do not appear on a dynamic slice, and, as such, are ineffectually covered. In Listing 1, $gap(CC_{DS})$ is empty, as all covered statements also appear on the dynamic slice.

C. Checked Coverage using Observational Slicing (CC_{OS})

Observational slicing is a language-independent technique that constructs a program slice through repeated line deletion and program behavior observation [25]. If the line can be deleted and the original behavior of the slicing criterion (i.e., the state of a variable at a given location) remains, the deletion is accepted. If the deletion causes a change in program behavior under the given slicing criterion, the deletion is rejected. Like dynamic slicing, observational slicing creates a slice that can be used to calculate checked coverage and determine whether program code influences test oracle outcomes.

Definition II.5 (Checked Coverage using Observational Slicing)**.** Checked Coverage using Observational Slicing requires each program line $l \in P$ to be on at least one observational backward slice from an explicit test suite check.

It is worth noting that an observational slice is calculated in terms of lines; however, the oracle gap refers to the statements on those lines. In terms of the $gap(CC_{OS})$, the effectual statements are those covered and appearing on lines on a slice (i.e., not deleted), whereas the ineffectual statements are those that are covered yet do not appear on a slice (i.e., deleted). The $gap(CC_{OS})$ comprises the originally covered, but collectively deletable lines (i.e, the set of statements on ineffectual lines).

D. Pseudo-tested statement identification (PTSI)

Where the test suite executes a production code element (e.g., statement or method), yet the same element can be deleted from the project without altering the test outcomes, this code is said to be "pseudo-tested" [13], [14]. When identifying pseudo-tested statements, code can be categorized as follows:

Definition II.6 (Pseudo-tested Statement)**.** Code that the tests execute, but is removable without changing test outcomes.

Definition II.7 (Required Statement)**.** Code that the test suite executes but is not removable without changing test outcomes.

The set of pseudo-tested statements form $gap(PTSI)$. PTSI differs from CC_{OS}, as CC_{OS} uses collective line deletions to form a slice, whereas PTSI uses only individual deletions or default values to evaluate single statements.

III. Evaluation

This section states and contextualizes the five research questions that this paper answers about oracle gap calculation.

As there is no existing comparable evidence, we must first quantify the oracle gaps calculated by each OGCA, enabling us to compare and contrast them in subsequent research questions. This includes the size of the gaps, and the similarity between them. A low similarity in gap content would suggest the techniques are not interchangeable, each having a unique use case. This directly leads to research questions 1 and 2:

RQ1: Prominence *– What proportion of a subject's production code lies within the oracle gap calculated by each approach?*

RQ2: Distribution *– How do the contents (i.e., program statements) of each approach's oracle gap differ and/or overlap?*

After understanding the breadth of the oracle gaps, we can look at the individual statements that are declared ineffectual by each OGCA. Analyzing the code statements and patterns identified by each OGCA is critical to evaluating their potential use cases. Furthermore, if developers use oracle gaps as an intermediate step towards, or instead of, mutation testing, a gap must effectively highlight priority areas needing further testing. To reflect this, we ask research questions 3 and 4:

RQ3: Categorization *– Which program statement types appear within the oracle gaps calculated by each approach?*

RQ4: Fault Detection *– How well does each oracle gap calculation approach highlight areas of low mutation score?*

Finally, as developers typically focus on a single class when writing unit tests, it is essential to be pragmatic regarding each approach's performance. As such, we ask research question 5:

RQ5: Performance *– How do the execution times of different oracle gap calculation approaches compare?*

Answering these questions will enhance understanding of the oracle gaps and how different approaches may be used.

A. Tooling

We use statement coverage and an oracle-based adequacy criterion to form an OGCA to identify an oracle gap. We limit the study to statements within methods to ensure comparability across tools, and, therefore, use PseudoSweep's statement coverage as the coverage tool to calculate oracle gaps [26]. To implement CC_{DS}, we used the Slicer4J dynamic slicer for Java, replicating other studies of checked coverage, to enable the analysis of projects using Java 8 or 9, and overcome some limitations of JavaSlicer [27], [28], [29]. For PTSI, we used PseudoSweep to identify stand-alone pseudo-tested statements within the study projects [26]. Finally, to evaluate CC_{OS}, we use the PORBS (Parallelized ORBS) Observational Slicer implementation [30]. We use PIT, a state-of-the-art Java mutation testing tool [31] to evaluate whether the oracle gaps reveal areas of low fault detection. We implemented a tool called GapGrep, as overviewed in Figure 1, to run each OGCA and report and analyze its output in a unified manner. We use GapGrep, available in the replication package [18], to calculate and analyze the oracle gaps and report the results.

B. Projects

Our targeted projects consisted of open-source Java projects compatible with our chosen OGCA tools. As such, each project must use Java 8 or 9 (Slicer4J and PseudoSweep), JUnit 4 or 5, and be a single-module Maven project (PseudoSweep). Projects should also avoid threading, which can lead to inaccurate results in PseudoSweep [26]. All selected projects must also compile and have a passing test suite.

To ensure this study evaluated a range of projects, we systematically selected projects from the Maven Central Repository, using the OSSF Criticality Score to gauge the most influential and important projects [32], [33]. For diversity in the project set, we picked the top five Java projects for each Criticality Score parameter (both positively and negatively weighted), and the highest GitHub Stars count. The parameters of the criticality score, using the original "Pike" configuration, include GitHub repository statistics such as how recently it has been maintained and its dependents count. This initial process yielded 70 projects, from the 12 categories containing five projects each. However, initial experiments surfaced OGCA execution time as a significant constraint, with experiment runs for some individual Java classes taking multiple days. To ensure a feasible project set, we randomly selected six projects from our initial list. From each project, we identified five class and unit test class pairs, resulting in 30 classes under test. We include a complete list of the classes studied in Table I. Of the projects selected, facebook-java-business-sdk obtained the highest criticality score, inutils4j had the highest contributor involvement, euclid was the least updated, eo-yaml had the highest user activity in the last 90 days, and finally java-string-similarity and tabula-java had the highest star counts.

C. Method

We used GapGrep in these steps to answer the five RQs.

1) Execute Tools: As shown in Figure 1, to calculate the oracle gaps for each of the three OGCAs, GapGrep takes PseudoSweep's statement coverage calculation (i.e., statement coverage in method bodies only) and identifies which covered statements do not fulfill each approach's oracle-assessment criterion. To ensure a feasible analysis, we focused solely on unit test classes directly targeting the corresponding class-under-test. For instance, our analysis of the Skip class only considered tests within the SkipTest class. Consequently, the coverage data for each class, as presented in Table I, only reflects statement coverage achieved by its dedicated test class.

With GapGrep, we run Slicer4J (CC_{DS}), PORBS (CC_{OS}), PseudoSweep (PTSI), and PIT (mutation testing) on each class and test-class pair. For CC_{DS}, we collate the dynamic slices obtained from each JUnit 4 or 5 assertion into a single list. For CC_{OS}, we collect the list of deleted lines from the class under test, and for PTSI, we collect the statement location, statement coverage, and the pseudo-tested statements list. Finally, after executing the test suite for each class, we take the list of mutations made to the program during mutation testing, their locations, and their status (i.e., uncovered, killed, or survived).

2) Calculate Oracle Gaps: Using GapGrep, we collate the slices and PseudoSweep's data to identify the ineffectually tested statements that make up the oracle gaps, as shown in Figure 1. For CC_{DS} and CC_{OS}, we take the respective program slices and identify the covered statements, according to PseudoSweep's statement coverage calculator, that are not on the slice. This forms each OGCA's oracle gap. PseudoSweep uses internal coverage calculation and thus does need additional tooling to calculate the PTSI oracle gap [26].

3) Quantify Overlaps and Differences in Oracle Gaps: We compare the sets of program statements in each of the three oracle gaps with Jaccard Similarity, a metric that can compute the similarity between two sets that may not be equal in size. Jaccard Similarity is calculated by counting the number of source code statements that occur in both sets and dividing by the count of items appearing in either set. Therefore, the Jaccard Similarity J for two sets of program statements, A and B, is calculated by $J(A, B) = |A \cap B|/|A \cup B|$. We use the Jaccard Similarity score to compare the set pairs of varying sizes to quantify the overlap between the different oracle gaps.

4) Characterize Gap by Statement Types: To contextualize the contents of each oracle gap, we performed a manual study of the statements within the union of the oracle gaps for six classes (one from each project under test). For this, we used a negotiated agreement approach as used in other software engineering studies [34], [35], [36]. Each author individually assessed the purpose of the statements within each oracle gap before coming together to discuss the assessments and arrive at a unanimous decision on the purpose of each statement. We then used the agreed-upon assessments to characterize the code patterns in each gap and highlight how each OCGA differs.

5) Calculate Mutation Scores for Each Oracle Gap: Using the PIT mutation tool for Java, with its STRONGER group of 13 mutation operators, we calculate the mutation score for each class, given its unit tests [37]. We then identify all the mutants, which line they occurred on, and whether they were killed, surviving, or uncovered. Using this, we could then identify the mutants placed in oracle gap statements and calculate the mutation score for each oracle gap for each class.

6) Compare Oracle Gap Calculation Performance: We calculate the execution time for each tool (including PIT) by timing how long it takes to execute on the class-under-test. For PseudoSweep, this is an overestimation as the run script includes the calculation of pseudo-tested statements, whereas for Slicer4J and PORBS, GapGrep calculates this afterwards.

D. Threats to Validity

External: This study leverages 30 open-source Java classes that fulfilled a range of criteria. Balancing feasibility for research purposes and practical use cases is a challenging task, and as such, computational expense necessarily restricted the number of projects and classes. Although this set of classes limited the study's breadth, it facilitated a deeper evaluation of the oracle gaps. Since our study was also restricted to Java projects, its results may not be generalized to other project sets using different languages, library versions, or threaded projects. Furthermore, our evaluation considered three oracle gap calculation approaches. Other strategies and implementations may yield varying results, and further studies should explore this. For mutation testing, we used the PIT mutation testing tool for Java and its STRONGER mutant set to generate and evaluate traditional mutants [37]. Yet, a different mutation testing tool or mutant set would have yielded different results.

Internal: We use Slicer4J, PORBS, and PseudoSweep to instantiate each OGCA. However, these implementations may not be error-free. Despite our best efforts, our implementation of checked coverage using Slicer4J and PORBS may contain defects that could impact the results. Both Slicer4J (CC_{DS}) and PseudoSweep (PTSI) manipulate the project-under-test by instrumenting source code (PseudoSweep) or moving test classes (Slicer4J), which could lead to an inaccurate oracle gap calculation if the original project behavior is not retained. Mistakes are also possible in GapGrep's amalgamation of tool outputs for evaluation. We, therefore, include tool code and additional scripts in the replication package for transparency, enabling researchers to replicate and extend this work [18].

Construct: The assumption that the mutation score is an indicator of test quality underpins RQ4 and the conclusions drawn from the results. This assumption is a notable threat to the construct validity of this study since there is conflicting evidence on this relationship. However, our choice is substantiated by studies demonstrating the correlation between mutation score and fault detection effectiveness [9], [10], [38]. Individual researchers' expertise may have biased the discussion within our manual study using negotiated agreement. We mitigated this by having each author reach individual conclusions before the discussion, where each author could justify their reasoning before coming to a collective decision.

IV. Results

A. RQ1: Prominence

Table I shows the distribution of covered, on-slice (CC_{DS} and CC_{OS}), and required lines (PTSI) within the project classes. For each technique, we identify the oracle gaps in each class. We first present the statement coverage data calculated using PseudoSweep for the class, given the paired unit test class. We use this to identify the effectual statements (i.e., those that are covered *and* appear on the slice), and the ineffectual statements (i.e., those that are covered but *do not* appear on the slice). The sum of effectual and ineffectual statements for each technique will always equal the total covered statements. The slices are calculated using lines rather than statements, but are paired with the appropriate statement to calculate oracle gaps. PTSI calculates required (i.e., effectual) statements differently, as a required statement must be covered by definition. Therefore, there is no separate slice counterpart for this technique. The following analysis of Table I presents the overall trends, followed by a deeper exploration of the prominence of oracle gaps highlighted in individual classes.

1) Prominence of Dynamic Slice Oracle Gap ($gap(\text{CC}_{DS})$): The $gap(\text{CC}_{DS})$ contains the statements covered under statement coverage, yet do not appear on a dynamic slice from a test assertion. Overall, 1201 statements appeared on at least one of Slicer4J's dynamic slices from the test class that corresponded with the class-under-test. Of the covered statements, 1021 also appeared on the dynamic slice and were therefore effectually covered. This left 459 (31%) statements that were ineffectually covered and therefore in $gap(\text{CC}_{DS})$.

Looking at the CC_{DS} data in Table I, we can see that On Slice does not necessarily mean covered. The statement coverage used only evaluates statements within methods; therefore, it will not account for elements on the slice beyond this. We can see that the number of effectual elements is greater for most classes than the number of ineffectual elements. For the 10 classes where the ineffectual count is greater than the effectual count there does not appear to be a correlation with the relative slice size or coverage. Notably, at least one such class appeared in each project-under-test. We also observe that MyReflectionUtils, HashedListAdaptor, and Event are the only classes that contain 0 effectual statements under CC_{DS}. Furthermore, despite 8 classes achieving 100% coverage, only 2 of them have no oracle gap (i.e., 0 ineffectual statements).

2) Prominence of Observational Slice Oracle Gap ($gap(CC_{OS})$): The observational slices are generated by observing the test classes' pass/fail behavior using PORBS.

TABLE I
THE COUNTS OF ELEMENTS FULFILLING EACH CRITERIA

	Statement Coverage			CC_{DS}			CC_{OS}			PTSI	
project / Class	#Stmts	#Cov	%Cov	On Slice	#Eff	#Ineff	On Slice	#Eff	#Ineff	#Eff	#Ineff
euclid / Real3Range	36	33	92	39	28	5	104	32	1	32	1
euclid / Int2Range	54	30	56	26	22	8	78	25	5	30	0
euclid / IntSquareMatrix	72	65	90	48	40	25	146	62	3	62	3
euclid / Line2	151	118	78	107	98	20	260	106	12	116	2
euclid / Transform2	213	90	42	50	38	52	173	58	32	86	4
inutils4j / MyNumberUtils	5	4	80	4	4	0	17	4	0	3	1
inutils4j / MyMapUtils	4	4	100	4	4	0	50	4	0	4	0
inutils4j / MyReflectionUtils	29	18	62	0	0	18	63	14	4	18	0
inutils4j / MyArrUtils	65	20	31	9	9	11	92	20	0	19	1
inutils4j / MyStringUtils	1025	476	46	388	384	92	1041	131	345	436	40
eo-yaml / JsonYamlDump	10	10	100	13	10	0	34	10	0	10	0
eo-yaml / ReadYamlStream	14	14	100	17	11	3	65	13	1	14	0
eo-yaml / WellIndented	26	21	81	3	2	19	57	19	2	21	0
eo-yaml / Skip	17	17	100	4	3	14	90	15	2	17	0
eo-yaml / ReadYamlMapping	68	56	82	74	51	5	265	53	3	56	0
facebook-java-business-sdk / HashedListAdaptor	34	26	76	1	0	26	70	24	2	26	0
facebook-java-business-sdk / BatchProcessor	12	12	100	3	3	9	116	10	2	12	0
facebook-java-business-sdk / CAPIGatewayEndpoint	38	17	45	17	15	2	92	12	5	17	0
facebook-java-business-sdk / ServerSideApiUtil	103	65	63	38	38	27	176	48	17	64	1
facebook-java-business-sdk / Event	313	85	27	153	85	0	1694	33	52	85	0
tabula-java / Cell	20	6	30	7	6	0	37	4	2	4	2
tabula-java / Line	28	19	68	12	10	9	61	18	1	19	0
tabula-java / Table	27	15	56	17	11	4	74	9	6	8	7
tabula-java / Rectangle	55	39	71	36	29	10	141	38	1	34	5
tabula-java / TextElement	116	80	69	37	34	46	173	56	24	80	0
java-string-similarity / LongestCommonSubsequence	22	20	91	10	9	11	45	12	8	20	0
java-string-similarity / SorensenDice	17	17	100	9	9	8	49	12	5	17	0
java-string-similarity / OptimalStringAlignment	26	26	100	14	11	15	50	16	10	26	0
java-string-similarity / RatcliffObershelp	34	34	100	21	21	13	68	30	4	34	0
java-string-similarity / NGram	48	43	90	40	36	7	52	9	34	43	0
Total	2682	1480	N/A	1201	1021	459	5433	897	583	1413	67

Suppose a test is unsuccessful (i.e., does not compile, fails, or it causes the throw of an unhandled exception) after deleting a line. In that case, the line is an observed dependency contributing to the test case passing. Overall, the $gap(\text{CC}_{OS})$ contained the most statements, at 583 ineffectual statements, accounting for 39% of covered statements. However, this trend does not represent the project set, as the 345 ineffectual statements in MyStringUtils bias this overall value. For all but five Java classes, the $gap(\text{CC}_{OS})$ was smaller than or equal to that of $gap(\text{CC}_{DS})$. Interestingly, the observational slice accounts for significantly more lines than are covered, highlighting an inefficiency when calculating oracle gaps. For example, in Real3Range, 33 statements are covered, yet 104 lines are on the slice. We can attribute this to how observational slicing constructs its slice, using observed dependencies, including any Java lines required for compilation and contributing to successful runs. In this sense, the observational slice is a more complete program slice; however, in this use case, not all of the slice provides relevant information when calculating a slice for an OGCA.

3) Prominence of Required Statements Oracle Gap (gap(PTSI)): The key difference between PTSI's required (i.e., effectual) statements and slices lies in how they contribute to test outcomes. Slices are statements that collectively cause a test to pass or fail, whereas, for required statements, we evaluate the individual impact on the test outcome. Across all statements, only 67 were pseudo-tested (i.e., ineffectual), far fewer than for the slicing-based techniques and just 5% of the covered statements. For all classes, there were more effectual statements than ineffectual, with only 11 classes having any ineffectual statements. MyStringUtils had 40 ineffectual statements, but its source code size is larger than the other classes.

Conclusion for RQ1. All OGCAs identified oracle gaps, but their sizes varied from 67–583 statements. The $gap(\text{CC}_{OS})$ often contained more sliced statements than covered statements, exposing computational inefficiency. The gap(PTSI) revealed a remarkably smaller gap of 67 statements, suggesting it is a less sensitive oracle-adequacy criterion.

B. RQ2: Distribution

To evaluate the distribution of the statements within the oracle gap, we calculate and present the Jaccard Similarity scores for each pair of oracle gaps. A Jaccard Similarity score of 1 indicates the elements of two sets are identical, whereas a score of 0 would indicate no common elements. In terms of oracle gap content between the different pairs of techniques, there are varying degrees of overlap. The $gap(\text{CC}_{OS})$ and $gap(\text{CC}_{DS})$ pair exhibit a similarity score of 0.21, which is a higher similarity than for any of the other pairs. Furthermore, the $gap(\text{CC}_{OS})$ and gap(PTSI) similarity is low at just 0.09, with $gap(\text{CC}_{DS})$ and gap(PTSI) being even lower at 0.06. The low similarity between gap(PTSI) and those of the other

TABLE II
STATEMENTS ARE CATEGORIZED THROUGH NEGOTIATED AGREEMENT. BOLD VALUES REPRESENT THE TECHNIQUE WITH THE HIGHEST COUNT.

Category		Total	CC$_{DS}$	CC$_{OS}$	PTSI
Checks	Special Case Check	7	**7**	2	0
	State Check	6	**6**	0	0
	Output Check	4	0	**4**	4
	Trivial Output Check	4	0	**4**	1
	Input Check	1	**1**	0	0
Update	Output Update	15	**15**	12	12
	State Update	6	**6**	0	0
Initialization	State Initialization	7	**6**	2	1
	Output Initialization	5	**5**	0	0
Return	Output Return	9	**6**	0	4
	Default Return	3	1	0	**2**
	Private Method Return	1	**1**	0	0
	Trivial Output Return	1	**1**	**1**	0
Other	Data Loading	322	68	**310**	8
	Iteration	25	**24**	2	1
	Defensive Programming	14	9	**11**	6
	String Processing	12	0	**12**	11
	Mathematical Computation	5	**5**	0	0
	Throw Exception	3	**3**	2	2
	Scheduling	2	**2**	**2**	0
	Parent Call	1	0	**1**	**1**

OGCAs is likely a reflection of the difference in statement set sizes. For instance, as noted in RQ1's response, gap(PTSI) contained 67 statements, compared to 583 in gap(CC$_{OS}$).

Conclusion for RQ2. The statement sets in the oracle gaps identified by each tool are dissimilar. The gap(CC$_{DS}$) and gap(CC$_{OS}$) pair are most similar (0.21), while CC$_{DS}$ and PTSI are least similar (0.06). This is likely a reflection of the statement set size difference between the oracle gaps.

C. RQ3: Categorization

Intuitively, the statements in the oracle gaps represent the program source code that, according to each OGCA, was not well tested. To better characterize the statements for which the tests fall short, the three authors performed a manual study of the purposes of the code in the oracle gaps, using negotiated agreement to focus on 454 statements across 6 classes. We present our categorization of the statements in Table II, along with examples to describe the code patterns in the oracle gaps. When possible, we preserve the original code and indentation, adjusting and contracting it for readability as needed.

1) Checks (26): We defined check statements to include code where a conditional is used to evaluate some condition to change the path of the program execution. Such ineffectual checks were identified across the oracle gap from each OGCA.

Special Cases were the most common checks to appear in the oracle gaps. These include code structures such as edge-case checks, complex-conditional checks, and equality checks.

State Checks included checking flags and markers to decide on the path through the program. WellIndented, for example, used these checks on ineffectual Lines 125 and 126 to find if an incorrect state had been entered and throw an exception.

```
if (!"...".equals(previous....()) && lineIndent >
 prevIndent) {
     if (!":".contains(prevLineLastChar)) {
```

We identified 4 statements as output checks, where the output was the method's return value. Each instance occurred in the MyStringUtils class, to iteratively perform string processing (and therefore also included in Section IV-C5), such as the following ineffectual while statement check on Line 1801.

```
while (text.contains("\b")) {
  text = text.replace("\b", " ");
}
```

Trivial Output Checks included code where, given the set of inputs, applying a given operation on them was redundant, so an early check was performed to avoid doing so. In the method "capitalize" in MyStringUtils, for example, if the first character was already a capital, it was redundant to capitalize it, so it returned the string without changes, as shown below:

```
if (Character.isUpperCase(c)) {
  return s;
}
```

The check itself was the ineffectual statement in this example.

We found an instance of an Input Check in MyStringUtils using a try statement that we could not classify under another category. The code used the ineffectual try statement to attempt to parse an input string as an integer, identified only by CC$_{DS}$.

2) Update (21): We designated updates as statements that reassigned existing variables, given a behavior in the program. In particular, we identified State Updates and Output Updates. State updates included class state or internal method state statements, such as updating a counter variable or boolean flags used to track state. All 6 were identified by CC$_{DS}$ as in the oracle gap. One example in the OptimalStringAlignment class tracked the cost while calculating a distance matrix.

```
cost = 1;
if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
    cost = 0;
}
```

Here, CC$_{DS}$ deemed both the cost variable updates ineffectual.

Output updates included reassignments to the variable being returned by the method. CC$_{DS}$ placed all 15 such statements in the gap, but CC$_{OS}$ and PTSI also identified 12 of these.

3) Initialization (12): Initializations include ineffectual variable declarations. State Initialization included variables that were declared with the sole purpose of tracking state. We found this to occur 7 times, with 6 being identified by Slicer4J, 2 by PORBS, and 1 by PseudoSweep. For instance, in the WellIndented class, Line 93 initialized a boolean flag to track "withinBlockScalar" throughout the method:

```
    boolean withinBlockScalar = false;
```

The oracle gaps also contained 5 Output Initializations that initialized the variable returned by the method. For example, the "lowerTriangle" method in the IntSquareMatrix class contained an ineffectual output initialization on Line 364, as the triangle variable is returned later in the method.

```
      IntArray triangle = new IntArray((n * (n + 1)) / 2);
```

4) Return (14): We used return as a category, but delved further into the purpose of the return statement. This sample had no void return statements, so all statements returned a variable or value. We found one instance where the return statement returned a Trivial Output that did not need further operations from the method. Line 21 in the Rectangle class shows where, as part of a comparator for ordering, an initial check returns early when the two inputs are equal.

```
        if (o1.equals(o2)) return 0;
```

We also found one instance of a return statement in a Private Method. We labeled this as such because it was related to why it was in the oracle gap. This example was in the OptimalStringAlignment class on Line 120, within an internal implementation of a minimum method as shown below:

```
    private static int min(
            final int a, final int b, final int c) {
        return Math.min(a, Math.min(b, c));
    }
```

The three Default Returns appeared when the program behavior had not met any previous checks and therefore reached a specified default return value. An example is Line 1981 in the "hasJapaneseCharacter" method of MyStringUtils, where after iterating through each character in a string, if none of the characters fulfill the check, the method returns false.

```
  public static boolean hasJapaneseCharacter(String str) {
    for (char c : str.toCharArray()) {
      if (JAPANESE_BLOCKS.contains(UnicodeBlock.of(c))) {
        return true;
      }
    }
    return false;
  }
```

The most prominent return type found in the oracle gap was the Output Return, returning the result computed by the method. There were 9 instances of this, identified between $gap(\mathrm{CC}_{DS})$ with 6 and $gap(\mathrm{PTSI})$ with 4. Below is an example from Line 371 in the IntSquareMatrix class, where the "lowerTriangle" method ineffectually returns said triangle.

```
        return triangle;
```

5) Other (384): The Other category contains the code descriptions that did not fall into clear groupings. We discuss the categories in order of total statements. The largest category in this group was the Data Loading category, containing 322 statements, with most being identified by the OGCA using PORBS, as shown in Table II. These statements primarily appeared in the MyStringUtils class, which loaded many strings into map structures, such as Line 158:

```
    escapeStrings.put("&amp;", new Character('\u0026'));
```

The next most prominent category under Other was Iteration, which included for loops, while loops, and other structures primarily for iteration. Although this category does not provide insight beyond the statement types, it was worth noting that they were revealed mainly by the CC_{DS} approach.

Interestingly, we found 14 Defensive Programming statements in the oracle gaps, with some being revealed by each OGCA. These statements evaluated and addressed inputs and states to ensure that unexpected exceptions would not arise later in the program. For example, the OptimalStringAlignment class throws a NullPointerException if String s1 is null:

```
        if (s1 == null) {
            throw new NullPointerException("...");
        }
```

String Processing accounted for 12 statements, none of which were identified by CC_{DS}. These included the following in the "removeAccents" method in the MyStringUtils class:

```
    s = s.replace((char) 0xE1, 'a');
```

In this instance, the deletion-based techniques of CC_{OS} and PTSI seem to be tailored for detecting string processing statements that may need further tests or assertions.

Mathematical Computation includes any statement used to perform a mathematical operation given some values. We found 5 instances of this in the IntSquareMatrix class, where the whole class is directed at mathematical operations on matrices. For example, the following three ineffectual lines are used as part of a transpose matrix operation and were only identified as being in the oracle gap calculated with Slicer4J.

```
                int t = flmat[i][j];
                flmat[i][j] = flmat[j][i];
                flmat[j][i] = t;
```

The smallest three categories in Other were Throw Exception, Scheduling, and Parent Call. The three Throw Exception statements were another example of cases where we could not assign a deeper purpose than the statement type. This does not include all ineffectual exception throws, just those that could not be attributed to another purpose. We found two examples of Scheduling statements that were only identified as being in the oracle gap by CC_{DS} and CC_{OS}. The PseudoSweep-based PTSI was likely unable to identify these due to its difficulties with threaded code. Although this study aimed to avoid threaded code, there may have been instances where our pattern matching method did not reveal it. We found one instance where a parent call was in the gap, highlighted by CC_{OS} and PTSI. For this, the parent call was for a custom-written equality check on Line 211 in IntSquareMatrix:

```
    public boolean isEqualTo(IntSquareMatrix r) {
        return super.isEqualTo((IntMatrix) r);
    }
```

This class did not invoke Java's built-in equality methods.

Conclusion for RQ3. The gaps created by the three OGCAs had statements from 21 different sub-categories in 5 different overarching categories. CC_{DS} revealed the most ineffectual checks, iterations and initializations, whereas CC_{OS} and PTSI highlighted ineffectual string manipulation, data-loading and defensive programming. The divide between dynamic slicing and the deletion-based approaches suggests the code-under-test should influence OGCA selection.

D. RQ4: Fault Detection

Figure 2 presents the mutation scores for all of the statements within each oracle gap for each class by OGCA. We see the oracle gaps for each class as calculated using CC_{DS},

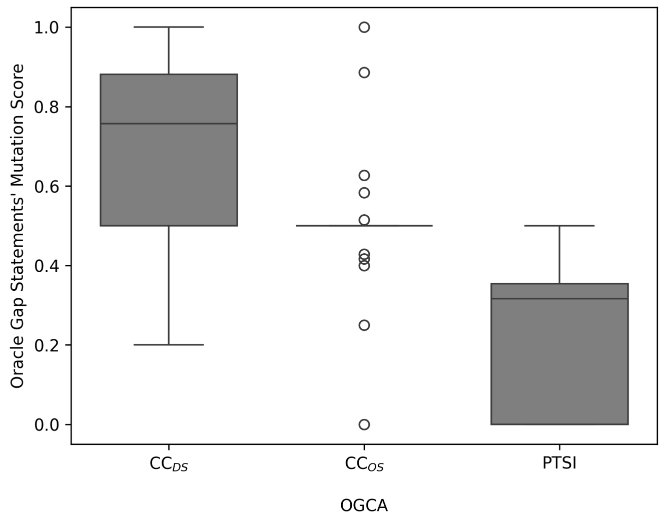

Fig. 2. A boxplot of the mutation scores for the statements in each oracle gap. Each box represents an interquartile range (IQR). The whiskers (i.e., the lines) extend to the max. and min. values but exclude outliers (i.e., the dots).

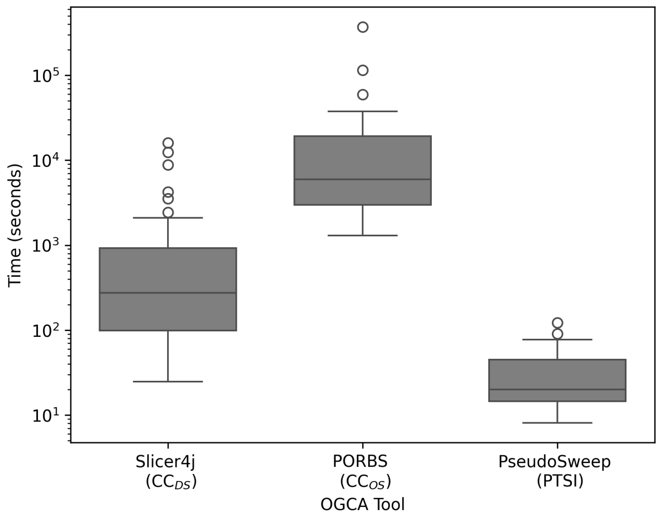

Fig. 3. A boxplot with a logarithmic scale showing the differences in execution times for the tools implementing CC$_{DS}$, CC$_{DS}$ and PTSI.

which has the highest median of mutation scores, with a wide range of scores from 0.20 to 1.00. This suggests that the approach may be less effective at highlighting under-tested areas where these faults may exist. In other words, this oracle gap identifies statements with high mutation scores that are, therefore, a lower priority for further testing. Ultimately, these high scores suggest that addressing the oracle gap identified by dynamic slicing may not help developers to reveal faults in their code. In contrast, PTSI's oracle gap of pseudo-tested statements achieves the lowest mutation scores with a median value of 0.32, indicating that it is the most effective of the three approaches at drawing attention to potential fault detection deficiencies. The mutation scores are generally low with little variation, showing a consistency in PTSI's pinpointing of tests that fall short by exhibiting weak fault detection. The mutation score of CC$_{OS}$'s oracle gap sits between the score of other two OGCA's gaps, with the smallest inter-quartile range of 0.00, causing no visible box in Figure 2. This means there is no variability in the middle 50% of the data; in this case, the mutation scores were 0.50. Yet, it has the most significant overall spread, with mutation scores at both 0.00 and 1.00. Although less effective than PTSI at highlighting areas of low fault detection, it is the most consistent of the OGCAs.

Conclusion for RQ4. The oracle gaps calculated by PTSI contain the lowest mutation scores, suggesting it is the most effective at revealing areas of fault detection deficiency.

E. RQ5: Performance

Figure 3 presents a box plot of each OGCA's execution times. As explained in Section III, the tools used by each OGCA are: Slicer4J for CC$_{DS}$; PORBS for CC$_{OS}$; and PseudoSweep for PTSI. Due to a significant time difference, Figure 3 uses a logarithmic scale. The Slicer4J execution times span from 24.8 seconds to 16067.0 seconds (approx. 4.5hrs). The box ranges from 99.3 seconds to 921.8 seconds (approx. 15 min), with no overlap with quartiles from other tool execution times. The Slicer4J execution times are generally higher than those of PseudoSweep, but mostly lower than PORBS. Slicer4J also shows more variability in execution time with some high outliers. Overall, PseudoSweep has the lowest execution times, ranging from 8.1 seconds to 121.7 seconds. This is the smallest range of any OGCA, as well as the least variability with an inter-quartile range of 30.2 seconds. We also noted that PseudoSweep had similarly quick execution times to PIT (i.e., the mutation testing tool used to answer RQ4), even though PseudoSweep is still a prototype tool.

PORBS' execution times are significantly higher than the other OGCAs, ranging from 1297.0 seconds (approx. 21min) to 371675.3 seconds (over 4 days) for a single Java class. This results suggests that using PORBS on an entire project becomes infeasible if it contains more than a few classes. The inter-quartile range of 16272.2 seconds (approx. 4.5hrs) for PORBS execution times also exposes its limited practicality.

Conclusion for RQ5. PORBS (CC$_{OS}$) was consistently impractically slow by a large margin, while Slicer4J (CC$_{DS}$) had varying execution times. Overall, PseudoSweep (PTSI) consistently had the fastest execution times per class.

F. Overall Conclusions

These results indicate that not all oracle gaps are equal, with differences in their size; the statements they contain; the relevance to fault detection; and the time to calculate, making it important for developers to make an informed choice of OGCA. Each covered statement is in the oracle gap because it does not cause a test to pass or fail, leaving the statement vulnerable to undetected incorrect behavior. This was echoed by the oracle gap mutation scores, demonstrating that the statements could contain many small syntactic changes (i.e., mutants) without causing any test failures, supporting the results of other recent studies [12], [13], [15]. PTSI's apparently more particular criterion for an ineffectual statement appears to be valuable in finding areas of low mutation score, whilst also benefiting from PseudoSweep's quicker runtime. CC_{DS} and CC_{OS} were less directed at areas of surviving mutants, and executed in substantially longer time periods, potentially wasting the already limited developer testing time. As such, this comparison yields valuable insights for developers when choosing an OGCA and the approaches' future development.

The results in Section IV-E are limited by the maturities of the tools involved. Continued optimization work may produce different execution times in the future. Further study should explore these oracle gaps with developers to understand how useful they find them in practice and whether they would take the time to add the relevant assertions. Developers could also inform automated solutions for addressing the oracle gaps.

V. Related Work

Oracle Gaps: The term oracle gap has been used by Jain et al. to refer to the value difference between coverage scores and mutation scores. While this provides a metric to evaluate a test suite, the metric alone does not provide actionable items, such as the priority statements necessitating improved testing that we evaluate in this study [39]. Hossain et al. found a statistically significant correlation between fault-detection effectiveness and host checked coverage gap size, demonstrating the importance of addressing the gaps between techniques. However, host checked coverage was fixed by the dynamic slicer, with a changeable coverage criterion and thus unsuitable for this study. Extreme mutation testing (XMT) has found pseudo-tested methods in prominent open-source Java projects, such as Apache Commons-Lang and Commons-Math [14]. These methods are areas of low mutation score, indicating fault detection weakness in their test suites [15]. As our study was not at the method level, XMT was not applicable to this study. Maton et al. extended this search for pseudo-testedness to the statement level, identifying methods that were "required" for the test suite to pass and could still contain pseudo-tested statements [13]. We employ their tool, PseudoSweep; however, we do not limit this study to only the required methods [26].

Slicing Approaches: Lee et al. introduced *Observation-based approximate dependency modeling* as a faster alternative to ORBS, producing an approximate slice [40], [41]. Although faster, the reduced accuracy of this approach is not appropriate for calculating checked coverage values. Lee et al. created a framework to reveal the causal dependence between program elements. This technique may be helpful in future work to infer the causes behind ineffectual elements [42]. Binkley et al. investigated "observational sensitivity to inadequate testing" where they used ORBs to slice on variables that may interest a maintainer [43]. We use PORBS to focus on the overall outcome of the test suite rather than value changes for individual variables or lines. DeMillo et al. introduced critical slicing as a fault localization approach that also uses deletion to create a program slice [44]. However, critical slicing is unsuitable for this study as the individual statement deletion checks and collective removal can produce incorrect slices.

Oracle-based Test Adequacy Techniques: Hossain and Dwyer surveyed oracle-based test adequacy metrics, highlighting the following as the prominent metrics in the area [45]. Beyond checked coverage, discussed in Section II-B, the two metrics they identified were state coverage and observable coverage. Koster and Kao developed state coverage to reveal whether unit tests evaluate program outputs and side-effects [46]. Their initial implementations were 70x slower than the standard JUnit test runner [47]. An extension on state coverage provided a generalized algorithm but did not provide enough feedback to help developers debug their code, as such, not making it valuable for this study [48]. Whalen et al. introduced observable MC/DC (OMC/DC) coverage to ensure that the effects of faults would be revealed by oracles [49]. By adding a path condition to MC/DC, OMC/DC requires stronger oracles in existing test cases rather than increasing the number of test cases. However, OMC/DC is not an oracle assessment technique, making it unsuitable for this study.

VI. Conclusions and Future Work

Calculating an oracle gap enables developers to pinpoint covered statements that do not cause an assertion to pass or fail, revealing vulnerabilities to incorrect behavior, and highlighting where testing is needed. Yet, with multiple OGCA options, developers cannot make an informed selection. This study quantitatively and qualitatively compared the oracle gaps of CC_{DS}, CC_{OS}, and PTSI across 30 Java classes in six projects, to assist developers in choosing a suitable OGCA.

Overall, each OGCA produced dissimilar oracle gaps in both statement set size and content. The set sizes ranged from 67–583 ineffectual statements, with the low Jaccard similarity scores (0.06–0.21) indicating that each OGCA identifies largely distinct sets of ineffectual code. Each OGCA identified different ineffectual statement types, with CC_{DS}'s gaps frequently involved iteration and update statements, while CC_{OS} and PTSI concentrated on data loading and string processing. Most notably, gap(PTSI) had the lowest mutation scores, making these statements high priority for improved fault detection. Furthermore, PseudoSweep for PTSI also emerged as the fastest OGCA tool. As such, this study identified PTSI as the most efficient and effective OGCA, enabling developers to quickly highlight the most vulnerable statements in need of additional tests and assertions, in the shortest execution time.

Given the promise of these results, we look to quantitatively broaden this study to complete Java projects and projects in other programming languages. Future work should also evaluate these OGCAs for specific coding styles, such as defensive programming, and gather developer insights to inform automated solutions for addressing oracle gap statement sets. Ultimately, the combination of this paper and the proposed future work will yield a practically useful way for developers to automatically identify how their test cases fall short.

Acknowledgments

Megan Maton is funded by the EPSRC Doctoral Training Partnership with the University of Sheffield, grant EP/W524360/1. Phil McMinn is supported, in part, by the EPSRC grant "Test FLARE" (EP/X024539/1).

References

[1] P. Straubinger and G. Fraser. A survey on what developers think about testing. In *Proceedings of the 34th International Symposium on Software Reliability Engineering*, 2023.

[2] M. Ivanković, G. Petrović, R. Just, and G. Fraser. Code coverage at Google. In *Proceedings of the 27th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[3] P. Piwowarski, Mi. Ohba, and J. Caruso. Coverage measurement experience during function test. In *Proceedings of the 15th International Conference on Software Engineering*, 1993.

[4] H. Hemmati. How effective are code coverage criteria? In *Proceedings of the International Conference on Software Quality, Reliability and Security*, 2015.

[5] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4), 1978.

[6] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5), 2011.

[7] A. V Pizzoleto, F. C Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software*, 157, 2019.

[8] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[9] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8), 2006.

[10] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the International Symposium on Foundations of Software Engineering*, 2014.

[11] L. Fernandes, M. Ribeiro, L. Carvalho, R. Gheyi, M. Mongiovi, Santos A. L. M., A Cavalcanti, F. C. Ferrari, and J C Maldonado. Avoiding useless mutants. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, 2017.

[12] S. B. Hossain, M. B. Dwyer, S. Elbaum, and A. Nguyen-Tuong. Measuring and mitigating gaps in structural testing. In *Proceedings of the International Conference on Software Engineering*, 2023.

[13] M. Maton, G. M. Kapfhammer, and P. McMinn. Exploring pseudo-testedness: Empirically evaluating extreme mutation testing at the statement level. In *Proceedings of the 40th International Conference on Software Maintenance and Evolution*, 2024.

[14] R. Niedermayr, E. Jürgen, and S. Wagner. Will my tests tell me if I break this code? In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, 2016.

[15] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. A comprehensive study of pseudo-tested methods. *Empirical Software Engineering*, 24(3), 2019.

[16] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry. Suggestions on test suite improvements with automatic infection and propagation analysis. In *arXiv:1909.04770*, 2019.

[17] M. Betka and S. Wagner. Towards practical application of mutation testing in industry — Traditional versus extreme mutation testing. *Journal of Software: Evolution and Process*, 34(11), 2022.

[18] Replication package (https://github.com/pseudotested/esem-2025-replication-package), 2025.

[19] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2016.

[20] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5), 2015.

[21] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 10th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2015.

[22] D Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, 2011.

[23] D. Schuler and A. Zeller. Checked coverage: An indicator for oracle quality. *Software Testing, Verification and Reliability*, 23(7), 2013.

[24] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3), 1988.

[25] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS: Language-independent program slicing. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, 2014.

[26] M. Maton, G. M. Kapfhammer, and P. McMinn. PseudoSweep: A pseudo-tested code identifier. In *Proceedings of the 40th International Conference on Software Maintenance and Evolution, Tool Track*, 2024.

[27] K. Ahmed, M. Lis, and J. Rubin. Slicer4J: A dynamic slicer for Java. In *Proceedings of the 29th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[28] JavaSlicer (https://github.com/backes/javaslicer), 2016 (Last Updated).

[29] R. Koitz-Hristov, L. Stracke, and F. Wotawa. Checked coverage for test suite reduction – is it worth the effort? In *Proceedings of the 3rd International Conference on Automation of Software Test*, 2022.

[30] S. Islam and D. Binkley. PORBS: A parallel observation-based slicer. In *Proceedings of the 24th International Conference on Program Comprehension*, 2016.

[31] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: A practical mutation testing tool for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016.

[32] OSSF. Criticality score v2.0.4 (https://openssf.org/projects/criticality-score/), 2024.

[33] Maven central repository: https://repo.maven.apache.org/maven2.

[34] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. *Sociological Methods & Research*, 42(3), 2013.

[35] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the 11th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2017.

[36] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. What do developer-repaired flaky tests tell us about the effectiveness of automated flaky test detection? In *Proceedings of the 3rd International Conference on Automation of Software Test*, 2022.

[37] PIT mutators (https://pitest.org/quickstart/mutators/), 2025.

[38] M. Papadakis, D. Shin, S. Yoo, and D-H. Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[39] K. Jain, G. T. Kalburgi, C. Le Goues, and A. Groce. Mind the gap: The difference between coverage and mutation score can guide testing efforts. In *Proceedings of the 34th International Symposium on Software Reliability Engineering*, 2023.

[40] S. Lee, D. Binkley, R. Feldt, N. Gold, and S. Yoo. MOAD: Modeling observation-based approximate dependency. In *Proceedings of the 19th International Working Conference on Source Code Analysis and Manipulation*, 2019.

[41] S. Lee, D. Binkley, R. Feldt, N. Gold, and S. Yoo. Observation-based approximate dependency modeling and its use for program slicing. *Journal of Systems and Software*, 179, 2021.

[42] S. Lee, D. Binkley, R. Feldt, N. Gold, and S. Yoo. Causal program dependence analysis. *Science of Computer Programming*, 240, 2025.

[43] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. ORBS and the limits of static slicing. In *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation*, 2015.

[44] R.A. DeMillo, H. Pan, and E.H. Spafford. Critical slicing for software fault localization. *Software Engineering Notes*, 21(3), 1996.

[45] S. B. Hossain and M. B. Dwyer. A brief survey on oracle-based test adequacy metrics. In *arXiv:2212.06118*, 2019.

[46] K. Koster and D. Kao. State coverage: A structural test adequacy criterion for behavior checking. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering*, 2007.

[47] K. Koster. A state coverage tool for JUnit. In *Companion of the 30th International Conference on Software Engineering*, 2008.

[48] D. Vanoverberghe, J. de Halleux, N. Tillmann, and F. Piessens. State coverage: Software validation metrics beyond code coverage. In *Proceedings of the International Conference on Current Trends in Theory and Practice of Computer Science*, 2012.

[49] M. Whalen, G. Gay, D. You, M. P. E. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *Proceedings of the 35th International Conference on Software Engineering*, 2013.